ABSTRACT

The DSP/BIOS real-time kernel is a scalable library that allows application developers to control how much of the real-time operating system is actually placed in target memory and to optimize the tradeoff between memory usage and functionality.

This application report contains information on the size impact of using DSP/BIOS modules in your application, as well as a number of hints and guidelines to help you optimize your application’s use of memory. It highlights different aspects of the DSP/BIOS library that are under your control and that impact the overall memory usage of the DSP/BIOS kernel.

As a DSP/BIOS user, you should be aware that it may be appropriate to change the way your application is configured over the life of the project. For example, host/target communication can be crucial during product development but may not be necessary when the application is deployed, allowing a reduction in the final overhead size. To that end, some hints given in this application report may not be applicable now, but may be relevant later.
1 DSP/BIOS Configuration Tool

Because DSPs are used in such a wide range of applications—from small battery operated devices to large multiprocessing systems—any general-purpose system software for DSPs must be extremely flexible in both functionality and size.

The DSP/BIOS kernel is a run-time library that provides a wide range of real-time kernel services broken up into different modules. Each of these modules can be linked in by a DSP application or left out entirely. The key to this is DSP/BIOS’s configuration capability. DSP/BIOS can be configured graphically or textually (using Tconf). Both work well inside Code Composer Studio. You can statically (that is, prior to run time) configure an application to link in only the services it actually needs. In this way, you can manage a memory-constrained application’s tradeoff between the functionality available through DSP/BIOS and the overall memory footprint of the application. The Configuration Tool allows you to configure different aspects of a particular module so that it is optimized to work within the application. Therefore, the general-use DSP/BIOS kernel can be transformed into an application-specific, run-time environment.

The tradeoffs between different DSP/BIOS modules and their impact on system memory can be complex because applying a module usually requires support of other modules. For example, the MBX module depends on (and therefore links in) the code from the SEM and QUE modules.

It is also important to realize that, even if a module’s code is linked in, it does not necessarily link in the entire module, but typically only the functions referenced by the application—an optimization that keeps the overall size impact of the DSP/BIOS kernel to a minimum.

Because of the complexity of these tradeoffs, it is important to understand that this application report does not set up an analytical model of estimating DSP/BIOS overhead, but rather gives some sizing scenarios based on a default configuration or a base configuration.
2 Creating a Minimal Footprint DSP/BIOS Application

This section contains hints on how to use the Configuration Tool to lower the overhead of the DSP/BIOS kernel. The actual size reduction of using these techniques depends on the specific application’s configuration.

2.1 Default Configuration

Unlike previous versions, the 5.xx version of DSP/BIOS provides a simple way to create a new configuration with minimal features enabled (that is, with Dynamic Memory Heaps, Real-Time Analysis, RTDX, and TSK Manager disabled). Even with all these features disabled, there are a few additional techniques that can be applied to further minimize DSP/BIOS code size usage.

The starting default configuration still includes message logging for execution tracking (LOG), built-in statistics gathering (STS), timer configuration and usage (CLK), interrupt vector handling (HWI), and basic system services (SYS) for halting and printing program execution.

All the values shown for the default configurations include code and data footprint sizing for the entire application. The application contains an empty main() function. The data values include the necessary C-initialized (.cinit) and uninitialized records (.bss) used by the application.

The default DSP/BIOS configuration file (*.tcf) calls the utils.loadPlatform() and the prog.gen() to generate the appropriate files as shown below:

```c
utils.loadPlatform("ti.platforms.dsk6416");

// !GRAPHICAL_CONFIG_TOOL_SCRIPT_INSERT_POINT!
if (config.hasReportedError == false) {
    prog.gen();
}
```

To create a default configuration yourself, you can use a similar script (as shown above) or have one created for you. To have one created in CCStudio, choose File->New->DSP/BIOS Configuration, select the platform you’d like to start from, and disable (un-check) features under the Enable DSP/BIOS Features as shown in the following figure:
The DSP/BIOS features you disable have the following functionality:

- **Dynamic Memory Heaps.** The DSP/BIOS kernel allows applications to dynamically allocate objects and storage areas in a heap area using the XXX_create() functions and the MEM_alloc(), MEM_calloc() and MEM_valloc() functions. Removing this functionality from the kernel allows the linker to remove the XXX_create functions and the MEM module code from the executable. The linker does not set up a memory heap area.

- **Real-Time Analysis.** The DSP/BIOS kernel provides the ability to extract real-time analysis (RTA) data from running applications. In this way, you can get visibility into performance, as well as insight into how the application runs on a per-thread and ISR basis. RTA data is extracted implicitly from a version of the DSP/BIOS library instrumented for this purpose and explicitly by developers who embed API calls in the application. The data is sent to the PC host when the RTA plug-ins running in CCStudio on the PC poll the target. This is done through background IDL threads that have the lowest priority in the DSP system. Although real-time analysis can be valuable (or even critical) during software development, it impacts the overall memory requirements to accommodate the instrumented DSP/BIOS library, as well as the RTA APIs and the host/target communication infrastructure.
• **RTDX.** The DSP/BIOS kernel includes Real-Time Data Exchange (RTDX) functionality. RTDX provides a mechanism for applications to move data between the DSP and host PC, without stopping the DSP and incurring only a minimal amount of intrusion. The RTDX library uses a scan-based emulator to move data via the JTAG interface.

• **TSK Manager.** The DSP/BIOS TSK manager allows applications to create and manipulate independent threads of control. Many DSP/BIOS applications may not take advantage of this at all, in favor of implementing run-to-completion threads using software interrupts (SWIs) or just implementing a single-threaded control loop. Disabling the TSK manager conserves system memory by eliminating unneeded DSP/BIOS code, as well as eliminating the need for a task stack to be allocated.

### 2.2 Base Configuration

Defining what constitutes a base DSP/BIOS setup is somewhat arbitrary. For the purpose of this document, a base configuration contains various techniques to minimize footprint of DSP/BIOS applications. The following Tconf configuration script illustrates the base setup as an example:

```c
utils.loadPlatform("ti.platforms.dsk6416");
/* Setting system stack size to 1k 8-bit bytes */
bios.MEM.STACKSIZE = 0x0400;
/* Disabling PRD and CLK */
bios.PRD.USECLK = 0;
bios.CLK.ENABLECLK = 0;
/* Disable the DSP/BIOS instrumented library */
bios.GBL.INSTRUMENTED = 0;
bios.GBL.ENABLEALLTRC = 0;
/* Removing SYS Handle functions */
bios.SYS.TRACESIZE = 0;
bios.SYS.ABORTFXN = prog.extern("UTL_halt");
bios.SYS.ERRORFXN = prog.extern("UTL_halt");
bios.SYS.EXITFXN = prog.extern("UTL_halt");
bios.SYS.PUTCFXN = prog.extern("FXN_F_nop");
/* Reducing the System LOG buffer */
bios.LOG_system.bufLen = 0;
// !GRAPHICAL_CONFIG_TOOL_SCRIPT_INSERT_POINT!
if (config.hasReportedError == false) {
    prog.gen();
}
```

The following subsections describe in detail each of the above statements in the configuration script. The base configuration, as in the default case, doesn’t make any DSP/BIOS API calls and contains an empty main() function only.

**NOTE:** Values for the system stack are configured to 1 K 8-bit bytes for C6000 platforms and 512 16-bit words on all other platforms. This maintains consistency across ISAs (instruction set architectures). On ‘C55x, the system stack and the stack both total 512 16-bit words.
2.2.1 Disabling CLK and PRD

The CLK module allows an application to invoke functions on a periodic basis. This module provides a real-time clock, and can be used in conjunction with the Real Time Analysis (RTA) tools to measure periods of time or to add a timestamp to event logs.

The Periodic module (PRD) allows periodic execution of program functions based on the CLK module. By default PRD is driven by a CLK instance. All PRDs run in the context of PRD_swi, which is, itself, a software interrupt (SWI).

By disabling the use of CLK to drive the PRD, the CLK object used by the PRD (PRD_clock) is removed. The SWI object is also removed when no PRD objects are configured.

If no other objects are dependent on CLK, the module can also be disabled. This brings significant savings since neither the SWI nor the CLK module are linked in (if no other SWIs are used). The following configuration statements were added to the default configuration for the base configuration:

```
bios.PRD.USECLK = 0; // remove dependency of PRD on CLK so CLK can be disabled
bios.CLK.ENABLECLK = 0;
```

Note that the TSK manager, when enabled, uses PRD to drive TSK ticks. The TSK ticks are responsible for advancing the system alarm clock to ready any TSK whose timeout interval has expired (for example, via TSK_sleep or SEM_pend). If the CLK module is not used to advance the system clock, the application may call TSK_tick to advance the system clock.

2.2.2 Disabling the DSP/BIOS Instrumented Library

The DSP/BIOS kernel instruments a few modules (TSK, SEM) to gather additional statistical information that can be seen with the CCStudio DSP/BIOS tools (DSP/BIOS -> Statistics View). By leveraging the non-instrumented DSP/BIOS library, code savings can be obtained in the case where the TSK or SEM module is being used.

Tracing by the TRC module is also disabled since RTA is not being used. This disables RTA tracing when the application is loaded. Tracing can be re-enabled through the RTA Control Panel in CCStudio or by calling TRC_enable in the application.

The following configuration statements were added to the default configuration for the base configuration:

```
bios.GBL.INSTRUMENTED = 0;   // Disable the instrumented kernel library
bios.GBL.ENABLEALLTRC = 0;   // Disable the TRC mask
```

NOTE: The footprint savings, when disabling the use of the DSP/BIOS instrumented library, is seen only when the TSK and SEM modules are being used in the application. Otherwise, either library has the same footprint impact. When using the non-instrumented version of the library, the generated DSP/BIOS linker command file (*.cfg.cmd) contains a reference to the bios_NONINST.a## instead of the bios.a## (## values are architecture dependent).
2.2.3 Removing SYS Handling Functions

The SYS module provides a set of basic system services such as halting program execution and printing formatted text. The SYS module is used by other DSP/BIOS modules in lieu of similar C library functions. The SYS module default handlers use DSP/BIOS UTL functions to perform these tasks.

Users can set their own functions for the SYS module to call. In this way, developers can control what handler functions are called by specifying their own custom handler functions, or specifying a smaller function such as the DSP/BIOS UTL_halt function.

The Configuration Tool controls the handler functions for the following SYS module functions:

- SYS_exit(), which is used for orderly program terminations.
- SYS_abort(), which is used to terminate the program from a catastrophic situation.
- SYS_error(), which is used to handle kernel and application error conditions.
- SYS_printf(), SYS_sprintf(), SYS_vprintf() and SYS_vsprintf(), which are used to output formatted data.

The DSP/BIOS kernel uses the SYS_error() function in the modules’ XXX_create() functions, in the MEM module functions, and by the SIO stream input and output manager, so an application taking advantage of these services should have some kind of SYS_error() handler in place.

Like all other functions, the SYS_printf() family of functions is linked into an application only if any of them are used. We recommend that applications avoid using the SYS_printf() family of functions entirely in favor of the LOG_printf() function, which is several orders of magnitude more efficient in terms of both memory usage and CPU cycles.

The following script lines were added to the default configuration to reduce the base configuration footprint:

```plaintext
bios.SYS.TRACESIZE = 0;                      // Reduce SYS buffer size to 0 (zero)
bios.SYS.ABORTFXN = prog.extern("UTL_halt"); // Set SYS functions to UTL_halt
bios.SYS.ERRORFXN = prog.extern("UTL_halt");
bios.SYS.EXITFXN = prog.extern("UTL_halt");
bios.SYS.PUTCFXN = prog.extern("FXN_F_nop");// Set the SYS putC function to a NOP
```

2.2.4 Reducing the System LOG Buffer

The LOG module is used by the DSP/BIOS system to log system execution information. The default configuration sets the LOG buffer size to 64 (words) of the LOG_system object. User LOG instances also default to 64 words (or 16 LOG records). Though LOG_system can’t be removed, the size of the LOG buffer can be modified. The following script line was added to reduce to LOG buffer size to zero (0).

```plaintext
bios.LOG_system.bufLen = 0;    // Set system LOG buffer to zero (0)
```

NOTE: Reducing the LOG_system buffer size provides only data savings. There is no code savings.
3 Module Sizing Applications

The applications described here are used to measure the sizing impact of adding various modules. The applications build upon each other. This means that the configuration modification and DSP/BIOS API calls used in one section are subsequently used in the following sections, unless otherwise specified.

The sizing information for DSP/BIOS 5.30 or greater is provided in the Results.htm file, which is installed with DSP/BIOS. These sizing values are also available on the DSP/BIOS download page (https://www-a.ti.com/downloads/sds_support/targetcontent/bios/index.html) for DSP/BIOS 5.20.05 or greater by following the DSP/BIOS Documentation → Results link. The sizing values contain information for all the following applications on all supported platforms.

The BIOS_INSTALL_DIR\packages\ti\bios\benchmarks\html\Results.html file in the 5.30 or greater DSP/BIOS release distribution contains links to HTML files that contain size information for the following applications on all supported platforms. All the values in the result tables are in 8-bit bytes. Though some platforms don't support 8-bit byte access, the values are displayed in this format to maintain consistency across architectures.

3.1 Base Application

This is the starting point for our measurements. The configuration used here is identical to the base configuration described in Section 2.2.

3.2 HWI Application

In the HWI application, support for the HWI dispatcher has been added. Enabling the dispatcher allows the application to call an Interrupt Service Routine (ISR) without needing to call other functions to save the register values during context switching. (ISR functions can be written in C.) Using the HWI dispatcher creates a one time code/data impact on the application. The dispatcher is enabled on a per-interrupt basis from the individual HWI property pages. The HWI Dispatcher is automatically linked into the application if RTDX is enabled or a CLK object is created.

To the base configuration script, the following has been added:

```c
bios.HWI.instance("HWI_INT8").useDispatcher = 1;
bios.HWI.instance("HWI_INT8").fxn = prog.extern("FXN_F_nop"); // Call to a NOP function
```

For the `C54x platform the following script lines were used:

```c
bios.HWI.instance("HWI_SINT4").useDispatcher = 1;
bios.HWI.instance("HWI_SINT4").fxn = prog.extern("FXN_F_nop");
```

The application also makes a call to the following DSP/BIOS API:

- `HWI_dispatchPlug()`
3.3 CLK Application

In the CLK application, the CLK manager is enabled and a CLK object has been added to the HWI application configuration. The CLK module is processor-dependent; therefore overhead values vary slightly between processors. The following configuration script lines were added:

```c
bios.CLK.ENABLECLK = 1;
bios.CLK.create("CLK0");
```

The application also makes a call to the following DSP/BIOS API:

- CLK_gethtime()

3.4 CLK Object Application

In the CLK object application, an additional CLK object has been created to illustrate the size impact of each CLK object. The following configuration script line was added to CLK application:

```c
bios.CLK.create("CLK1");
```

3.5 SWI Application

In the SWI application, a single SWI object has been added to the CLK object application configuration. The following configuration script lines were added:

```c
bios.SWI.create("SWI1");
bios.SWI.instance("SWI1").priority = 1;
```

The application also makes a call to the following DSP/BIOS API:

- SWI_post()

3.6 SWI Object Application

In the SWI object application, an additional SWI object has been created to illustrate the size impact of each SWI object. The following configuration script lines were added to SWI application:

```c
bios.SWI.create("SWI2");
bios.SWI.instance("SWI2").priority = 1;
```

3.7 PRD Application

In the PRD application, the PRD module was enabled to use the CLK module. A PRD object has also been added to the SWI object application configuration. The following configuration script lines were added:

```c
bios.PRD.USECLK = 1;
bios.PRD.create("PRD0");
```
Note: If a function executes at the frequency of the timer interrupt, it may be more efficient to create a CLK object to run the function rather than a PRD object. The reason for this is that CLK objects are implemented to run off the timer's hardware interrupt, while PRD objects are implemented as software interrupts (SWI). The caveat to this is that PRD objects cannot be replaced with CLK objects in all circumstances—for example, when the execution time of the function being called is long (that is, greater than or equal to half the timer interrupt rate).

The application also makes a call to the following DSP/BIOS API:

- PRD_getticks()

### 3.8 PRD Object Application

In the PRD object application, an additional PRD object has been created to illustrate the size impact of each PRD object. The following configuration script line was added to PRD application:

```plaintext
bios.PRD.create("PRD1");
```

### 3.9 TSK Application

In the TSK application, the TSK module was enabled and a TSK object has been added to the PRD object application configuration. Enabling the TSK module creates an IDL task object. All TSK objects created statically in this application are set to a stack size of 512 bytes. (Though some platforms don't support 8-bit byte access, the application is configured accordingly to maintain size consistency across architectures.) The TSK stack sizes for each TSK can be adjusted accordingly (see the TSK Module section in the DSP/BIOS API Reference for more information). The following configuration script lines were added:

```plaintext
bios.TSK.ENABLETSK = 1; // Enable the TSK module

/* Set TSK stacks of 512 (8-bit) bytes */
bios.TSK.instance("TSK_idle").stackSize = 0x100; // config is in (16-bit) words
bios.TSK.create("TSK0");
bios.TSK.instance("TSK0").stackSize = 0x100; // config is in (16-bit) words
```

For 'C6000 platforms, the following script lines were used:

```plaintext
bios.TSK.ENABLETSK = 1; // Enable the TSK module

/* Set TSK stacks of 512 (8-bit) bytes */
bios.TSK.instance("TSK_idle").stackSize = 0x200; // config is in (8-bit) bytes
bios.TSK.create("TSK0");
bios.TSK.instance("TSK0").stackSize = 0x200; // config is in (8-bit) bytes
```
For ‘C55x platforms the following script lines were used:

```c
bios.TSK.ENABLETSK = 1; // Enable the TSK module
/* Set sum of TSK stack and sysstack sizes to 512 (8-bit bytes) */
bios.TSK.instance("TSK_idle").stackSize = 0x0c0; // config is in (16-bit) words
bios.TSK.instance("TSK_idle").sysStackSize = 0x040; // config is in (16-bit) words
bios.TSK.create("TSK0");
bios.TSK.instance("TSK0").stackSize = 0x0c0; // config is in (16-bit) words
bios.TSK.instance("TSK0").sysStackSize = 0x040; // config is in (16-bit) words
```

The application also makes a call to the following DSP/BIOS API:

- **TSK_yield()**

### 3.10 TSK Object Application

In the TSK object application, an additional TSK object has been created to illustrate the size impact of each TSK object. The following configuration script lines were added to the TSK application:

```c
/* Set TSK stack of 512 (8-bit) bytes */
bios.TSK.create("TSK1");
bios.TSK.instance("TSK1").stackSize = 0x100; // config is in (16-bit) words
```

For C6000 platforms, the following script lines were used:

```c
/* Set TSK stack of 512 (8-bit) bytes */
bios.TSK.create("TSK1");
bios.TSK.instance("TSK1").stackSize = 0x200; // config is in (8-bit) bytes
```

For ‘C55x platforms, the following script lines were used:

```c
/* Set sum of TSK stack and sysstack sizes to 512 (8-bit bytes) */
bios.TSK.create("TSK1");
bios.TSK.instance("TSK1").stackSize = 0x0c0; // config is in (16-bit) words
bios.TSK.instance("TSK1").sysStackSize = 0x040; // config is in (16-bit) words
```

### 3.11 SEM Application

In the SEM application, a SEM object has been added to the TSK object application configuration. The following configuration script line was added:

```c
bios.SEM.create("SEM0");
```

The application also makes a call to the following DSP/BIOS APIs:

- **SEM_post()**
- **SEM_pend()**
3.12 SEM Object Application

In the SEM object application, an additional SEM object has been created to illustrate the size impact of each SEM object. The following configuration script line was added to SEM application:

```c
bios SEM.create("SEM1");
```

3.13 MEM Application

In the MEM application, the configuration is enabled to use memory heaps and a heap of 4 KB was added to the SEM object application configuration. (Though some platforms don't support 8-bit byte access, the application is configured accordingly to maintain size consistency across architectures.) The following configuration script lines were added:

```c
bios.MEM.NOMEMORYHEAPS = 0; //Enable Memory Heap usage
bios.MEM.instance("IRAM").createHeap = 1; // Create a Memory Heap in IRAM
bios.MEM.instance("IRAM").heapSize = 0x1000; // Set size of Memory Heap
bios.MEM.BIOSOBJSEG = prog.get("IRAM"); // Set BIOS Objects to IRAM heap
bios.MEM.MALLOCSEG = prog.get("IRAM"); // Set malloc/free to IRAM heap
bios.TSK.STACKSEG = prog.get("IRAM"); // Set dynamic TSK stacks to IRAM heap
```

For 'C54x, 'C55x, and 'C28x, the heap values in the configuration file are specified in 16-bit words as follows:

```c
bios.MEM.instance("DARAM").heapSize = 0x0800; // Values are in 16-bit words
```

The application also makes a call to the following DSP/BIOS APIs:

- MEM_alloc()
- MEM_free()

3.14 Dynamic TSK application

In the dynamic TSK application, DSP/BIOS run-time APIs create and delete a TSK. Creating a TSK dynamically (that is, at run-time) doesn’t require any changes to the configuration; it uses the same configuration as the MEM application. The only dependency, when using dynamically-created objects, is that a heap must be configured in order to successfully create the TSK object. The application makes a call to the following DSP/BIOS APIs:

- TSK_create()
- TSK_delete()

NOTE: Using the Configuration Tool to create static objects at design-time rather than dynamic objects at run-time allows the application to forego linking in the object creation and deletion APIs, and will also save the overhead of embedding these calls in your initialization and termination routines.
3.15 Dynamic SEM application

In the dynamic SEM application, DSP/BIOS run-time APIs are called to create and delete a SEM. As with the dynamic TSK application, no changes were required to the configuration. The application used the dynamic TSK application configuration and makes an additional call to the following DSP/BIOS APIs:

- SEM_create()
- SEM_delete()

3.16 RTA Application

In the RTA application, all real-time analysis features are enabled, including the use of RTDX and the DSP/BIOS instrumented kernel library. The system LOG buffer size is set to 1 KB. The following configuration script lines were added to the MEM application configuration:

```plaintext
bios.GBL.ENABLEINST = 1; // Enable Real-time Analysis
bios.GBL.INSTRUMENTED = 1; // Enable the use of the BIOS instrumented kernel
bios.GBL.ENABLEALLTRC = 1; // Enable all TRC mask logging
bios.enableRtdx(prog); // Built-in function to enable RTDX if available on the platform
bios.LOG_system.bufLen = 1024; // Set system LOG buffer to 1k
```

4 Module Dependency Table

The table below shows the module dependencies for each of the previously described applications.

<table>
<thead>
<tr>
<th>Module/Instance</th>
<th>Dependent Modules</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>HWI application</td>
<td>HWI, SWI</td>
<td>SWI is used when HWI dispatcher is enabled. One time code and data impact when using the HWI dispatcher.</td>
</tr>
<tr>
<td>CLK &amp; CLK Object</td>
<td>HWI, SWI</td>
<td>CLK size is processor-dependent (values may vary by processor). CLK objects require the use of the HWI dispatcher.</td>
</tr>
<tr>
<td>Application</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWI &amp; SWI Object</td>
<td>STS</td>
<td>STS is used in SWI statistics gathering.</td>
</tr>
<tr>
<td>Application</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRD Application</td>
<td>SWI, STS, CLK (depends on configuration)</td>
<td>PRD uses the CLK module by default (may be changed in the configuration). STS is used in PRD for statistics gathering.</td>
</tr>
<tr>
<td>TSK &amp; TSK Object</td>
<td>SWI, STS (not used if non-instrumented kernel is selected), SEM (if TSK_sleep is called)</td>
<td>Enabling module creates a TSK_idle function. TSK_idle stack is set to 512 bytes. TSK stacks are set to 512 bytes.</td>
</tr>
</tbody>
</table>
### Module/Instance Dependents

<table>
<thead>
<tr>
<th>Module/Instance</th>
<th>Dependent Modules</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEM &amp; SEM Object</td>
<td>QUE</td>
<td></td>
</tr>
<tr>
<td>Application MEM App.</td>
<td>None</td>
<td>Requires a heap.</td>
</tr>
<tr>
<td>Dynamic TSK</td>
<td>MEM</td>
<td>Requires a MEM heap to create the object.</td>
</tr>
<tr>
<td>Dynamic SEM</td>
<td>MEM</td>
<td>Requires a MEM heap to create to object.</td>
</tr>
<tr>
<td>Application RTA</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

## 5 Measuring DSP/BIOS Footprint for Custom Applications

This section explains how we measured the footprint of the sample applications described in Section 3 using a suite of automated tools.

### 5.1 Using the Linker Map or XML Listing

As of CGT version >= 5.0 (or CGT 3.x for ‘C55x) the code generation tools can create an XML listing file at link time in addition to a map file. We highlight this since parsing XML is far more robust than parsing the text-based map file.

The `-m` linker option generates a map file. The `-xml_link_info` option generates an equivalent representation in XML. In CCStudio, you can select this via Project–>Options ->Linker tab -> Advanced and filling in the fields for the XML Link info file.

Both the map and the XML file can be used to see how each memory section is allocated, and which software components are using them.
5.2 Using Object File Display Utility and CG_XML Scripts

The Object File Display utility (ofdxx.exe) can be used to view the content of object files, executables, and/or archive libraries in human readable or XML format. The utility is included with the Code Composer Studio code generation tools. It can be invoked from the command prompt (DOS) shell.

```
Command Prompt
C:\CCStudio_v3.1\6000\cgtools\bin>ofdxx -h
TMS320C6x Object File Display v5.1.0
Tools Copyright © 1996-2005 Texas Instruments Incorporated
Usage: ofdxx [-options] filename
Use 'ofdxx -h <option>' for more information on a particular option. Use 'ofdxx -h <text>' to search options for information regarding <text>.
Option arguments surrounded by [] declare they are optional. Option arguments separated by commas specify that the option's argument must be chosen from those arguments listed.
General Options:
   -g Append DWARF information to file information.
   -h Display help
   -o=file Output object file information to <file>.
   -x Output information in XML form.
```

- Several Perl scripts have been developed by TI, and are delivered in a package called cg_xml. These scripts do useful things with the XML output such as generating boot images, etc. The scripts, in particular sectti.pl, were used to generate most of the sizing data here. This and other useful code generation XML parsing scripts are provided at the following web site: https://www-a.ti.com/downloads/sds_support/applications_packages/index.htm

- The sectti.pl prints the name, size, and base address of each section in an object (.obj), executable (.out), or library (.lib) file. The output file can be either a text format or a CSV (comma separated values). For example:
Table 2. Summary of Linker Section Names

<table>
<thead>
<tr>
<th>Section Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.args</td>
<td>Arguments passed to main() (i.e., argc, argv and envp)</td>
</tr>
<tr>
<td>.bios</td>
<td>DSP/Bios code</td>
</tr>
<tr>
<td>.bss</td>
<td>Global and static variables</td>
</tr>
<tr>
<td>.cinit</td>
<td>Initialization tables for global and static variables</td>
</tr>
<tr>
<td>.const</td>
<td>Explicitly initialized global and static constant variables and string literals</td>
</tr>
<tr>
<td>.hwivc</td>
<td>Hardware interrupt vector table</td>
</tr>
<tr>
<td>.module</td>
<td>DSP/Bios module objects (.clk, .hst, .idl, etc.)</td>
</tr>
<tr>
<td>.module*</td>
<td>Data associated with a DSP/Bios module object</td>
</tr>
<tr>
<td>.pinit</td>
<td>Table of initialization functions</td>
</tr>
<tr>
<td>.stack</td>
<td>Boot stack</td>
</tr>
<tr>
<td>.switch</td>
<td>C switch statement tables</td>
</tr>
</tbody>
</table>
## 6 Conclusion

The sizing information for DSP/BIOS 5.30 or greater is provided in the Results.htm file, which is installed with DSP/BIOS. These sizing values are also available on the DSP/BIOS download page (https://www-a.ti.com/downloads/sds_support/targetcontent/bios/index.html) for DSP/BIOS 5.20.05 or greater by following the DSP/BIOS Documentation→Results link. The sizing values contain information for all the following applications on all supported platforms.

Size data is provided in the HTML files for the applications described in this document. This information provides an estimate of the DSP/BIOS footprint for a few sample applications. Since each application is unique, techniques described in Section 5 can be used to determine the exact footprint impact.

## 7 References

2. *TMS320C28x DSP/BIOS API Reference Guide* (SPRU625)
3. *TMS320C5000 DSP/BIOS API Reference Guide* (SPRU404)
IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI’s standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

<table>
<thead>
<tr>
<th>Products</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amplifiers</td>
<td>Audio</td>
</tr>
<tr>
<td>Data Converters</td>
<td>Automotive</td>
</tr>
<tr>
<td>DSP</td>
<td>Broadband</td>
</tr>
<tr>
<td>Interface</td>
<td>Digital Control</td>
</tr>
<tr>
<td>Logic</td>
<td>Military</td>
</tr>
<tr>
<td>Power Mgmt</td>
<td>Optical Networking</td>
</tr>
<tr>
<td>Microcontrollers</td>
<td>Security</td>
</tr>
<tr>
<td>Low Power Wireless</td>
<td>Telephony</td>
</tr>
<tr>
<td></td>
<td>Video &amp; Imaging</td>
</tr>
<tr>
<td></td>
<td>Wireless</td>
</tr>
</tbody>
</table>

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Copyright © 2006, Texas Instruments Incorporated